Overlay-Centric Load Balancing: Applications to UTS and B&B

Trong-Tuan Vu, Bilel Derbel, Ali Asim, Ahcène Bendjoudi, Nouredine Melab

Overlay-Centric Load Balancing: Applications to UTS and B&B

Trong-Tuan Vu* Bilel Derbel*, Asim Ali*, Ahcène Benjoudi† and Nouredine Melab*
*INRIA Lille Nord Europe, France
LIFL – Université Lille 1
{trong-tuan.vu,ali.asim}@inria.fr, {bilel.derbel,nouredine.melab}@lifl.fr
† CERIST, Algiers, Algeria
ahcene.bendjoudi@gmail.com

***Abstract*—To deal with dynamic load balancing in large scale distributed systems, we propose to organize computing resources following a logical peer-to-peer overlay and to distribute the load according to the so-defined overlay. We use a tree as a logical structure connecting distributed nodes and we balance the load according to the size of induced subtrees. We conduct extensive experiments involving up to 1000 computing cores and provide a throughout analysis of different properties of our generic approach for two different applications, namely, the standard Unbalanced Tree Search and the more challenging parallel Branch-and-Bound algorithm. Substantial improvements are reported in comparison with the classical random work stealing and two finely tuned application specific strategies taken from the literature.**

I. Introduction

A. Motivations and Goals

In this paper, we jointly address two challenging issues at the crossroads of two scientific computing fields: (i) designing generic distributed dynamic load balancing protocols for computing intensive applications and experimentally assessing their performance, and (ii) designing effective parallel algorithms for solving hard combinatorial optimization problems.

On one side, more and more computing resources are nowadays available in the form of aggregated clusters, grids, and personal computers scattered over possibly large scale distributed platforms connected via a network. This huge amount of computational resources offers an impressive computing power which is in theory sufficient to tackle many computing intensive problems. However, achieving high performance and scalability on large scale distributed systems is a difficult task especially when parallelizing highly irregular applications producing, dynamically at runtime, a set of unbalanced tasks. For such applications, no knowledge about the number/location of tasks generated over all execution time, neither their relative computing complexity nor difficulty can be assumed initially. Thus, load balancing strategies in which parallelism unfolds dynamically throughout the application execution greatly contributes in gaining substantial speed-up.

On the other side, many real-life problems coming from various domains, e.g., bio-informatics, logistics, health sciences, business intelligence, etc, are reputed to be hard combinatorial optimization problems. Many search and optimization algorithms for solving such problems require the enumeration of a huge space which is further of unpredictable structure. Consequently, sequential approaches often fail providing good performances in a reasonable time. Within this context, parallel and distributed computing are essential, not to say the key to efficiently solve the underlying computing challenges.

In particular, the well studied Branch-and-Bound (B&B) technique can be essentially viewed as a specific traversal of a tree representing a candidate solution space. Roughly speaking, branching in B&B allows one to decide which branch of the tree to explore next, and bounding allows one to decide whether to continue the exploration or to discard a whole part of the tree. Although B&B can be endowed with sophisticated mechanisms allowing to significantly reduce the number of branches to be explored, the size of the remaining tree is still huge when tackling large instances, thus making its sequential traversal very time consuming. More importantly, a B&B tree is unpredictable and extremely imbalanced. In fact, the variance in the size of the B&B subtrees is typically very high making it very difficult to evaluate the required computation for different subtrees to be traversed. Thus, besides B&B specific technicalities which are difficult to design and implement, B&B is extremely difficult to parallelize while achieving high performance and scalability. This is why effective and efficient dynamic load balancing strategies are important.

Designing efficient load balancing protocols is in fact a challenging research issue which is being actively studied for different computing intensive applications and in different distributed contexts such as distributed memory architectures, clusters, grids, and more generally global distributed systems harnessing a huge amount of computing resources. In this context, work stealing can be considered as a corner stone in most existing approaches. Work stealing is based on a very simple idea that idle processes (thieves) should attempt to acquire work by stealing it from other processes (victims), thus allowing workload to be transferred dynamically at runtime. However, considering a fully distributed environment, it is well understood that for such an approach to reach scalability and high performance, different interdependent tasks have to be carefully mixed and tuned, e.g., where to find work while avoiding synchronization delays, how to detect termination

without disturbing computations, what is the amount of work to transfer to distribute the load evenly.

In this paper, we describe a new generic message passing approach dealing with the previous issues. With respect to previous works, we show that our approach is extremely competitive to: (i) efficiently parallelize the state-of-the-art adversary benchmark for dynamic load balancing, and also (ii) to solve challenging combinatorial optimization problems using B&B. In the following, we describe in more details our contributions and results. Afterword, a throughout discussion on related work and techniques is given.

B. Contribution Overview and Results Summary

We consider the idea of dynamically balancing the load by structuring computing nodes in a peer-to-peer like topology, namely a tree, and distributing work according to the so-defined overlay. More precisely, we propose to dynamically adjust the amount of work transferred from a node to another according to the size of overlay subtrees. In other words, we make work flow over tree paths and we cooperatively balance the load according to the size of each peer subtree. To avoid that computing nodes stay idle for a long time, we further consider to extend the tree overlay with few long links in order to speed up work flow, thus reducing delays and gaining in parallel efficiency. The main idea behind our protocol is based on the simple observation that idle nodes should not be selfish when searching for work, but should acquire enough work to serve their neighboring peers. To study the properties of this overlay-centric approach, we conduct extensive experiments using two different applications, three highly competitive approaches taken from the literature, and up to 1000 real computing cores.

As parallel applications, we consider the Unbalanced Tree Search (UTS) benchmark and the challenging B&B algorithm. UTS was specifically designed to be a reference adversary for dynamic load balancing strategies [19]. The results obtained with UTS aims at proving that our approach is fully generic and can be used to effectively tackle a wide range of highly irregular applications. As discussed previously, Parallel B&B is such a challenging application which has been addressed by its own as a whole piece of research [5], [12], [14], [21].

To evaluate the relative performance of our approach, we implemented the generic Random Work Stealing (RWS) which is the state-of-the-art reference protocol in dynamic load-balancing. We further consider two specific B&B distributed approaches: the Master-Worker (MW) approach described in [17], and the Adaptive Hierarchical Master Worker approach (AHMW) recently studied in [2], [3]. For clarity, the detailed description of these two approaches is delayed to later in the paper. One should notice that MW is an extremely competitive B&B-specific approach, which was used in the past to solve for the first time a 22-years unsolved hard combinatorial optimization problem instance. The general idea of AHMW is similar to our approach in the sense that it explicitly organizes masters and workers in a multi-level tree overlay, and adapts the B&B load according to masters hierarchy.

Through extensive experiments, and using up to 1000 real cores of the Grid'5000 clusters [13], we report the relative performance of our approach under different overlay configurations and different scales. We show that substantial improvements are obtained for both UTS and B&B while compared with RWS and application specific schemes (MW and AHMW). For instance, while the parallel efficiency of our approach for UTS (96%) is comparable to RWS at the low scales, it scales significantly better achieving 77% against 64% for RWS at the higher scales of 512 cores. For B&B, it is even more impressive since we achieve up to 96% of parallel efficiency against less than 70% for RWS in the scale of 1000 cores. Specific to B&B, we consider ten different instances of the well known Flowshop problem and solve them using our approach, AHMW and MW. Using 200 cores, our approach is substantially better than AHMW (10 times faster overall). Compared to MW, our approach performs significantly better on 7 out of 10 instances, and slightly worse on the three others. But, when scaling the network up to 1000 cores, our approach continues scaling smoothly, while the performance of MW deteriorates and the execution time is even getting worse.

C. Related works and discussion

1) Work stealing: From the dynamic load balancing side, work-stealing is a reference approach among the particularly rich literature [4], [6], especially in the shared memory computing model [9], [19], [20]. In random work stealing, an idle node tries to steal work from the queue of other processes selected uniformly at random. This very simple approach is actually proved to have optimal performances under some circumstances [4]. At the price of a very fine tuning, random work stealing can be extended to the distributed memory context, e.g., [15], [8], [20]. However, it is generally known that straightforward extensions are not well suited to optimally exploit network resources of possibly many heterogenous platforms forming giant distributed systems, e.g., global infrastructures, large scale peer-to-peer networks, etc. This is mainly due to the high communication cost raised by the nature of such computing environments, e.g., inter-node communication, network bandwidth, heterogeneity, etc. ATLAS [1] and SATIN [26] are such two frameworks providing global load balancing based on hierarchical work-stealing. To the best of our knowledge, although existing hierarchical frameworks implicitly induce a multi level tree, we did not find any specific study on how tree overlay properties can be used to decide on the amount of work to be distributed.

The only work explicitly embedding an overlay structure in work stealing was conducted very recently in [24]. The authors therein use a hyper-cube as a lifeline graph to minimize missed stealings, thus allowing random work stealing to scale better. The authors implemented their approach using X10 programming language and validate it on two supercomputers, namely, a Blue Gene/P and a Power-7 infiniband cluster. When using 128 cores of a smaller x86 cluster, comparable to the ones used in this paper, they report 94% of parallel efficiency with the UTS benchmark. At the same scale we report 96%

of parallel efficiency for UTS and we further push our study at larger scales and for the more challenging parallel B&B application as will be discussed later.

Very recently, several hybrid and hierarchical approaches exploring the locality and the heterogeneity of hardware are being actively developed, e.g., [18], [22], [23]. To the best of our knowledge, most approaches dealing with the heterogeneity of hardware are essentially exploring the following idea. Basically, their proposals define a hardware hierarchy for stealing work in order to get benefits from the multi-level architecture of computing nodes. Typically, a random stealing policy is performed from the lowest to the highest one based on the hardware hierarchy, e.g., inside a multi-core node, across multi-core nodes. The amount of work (granularity) to be transferred from a node to another can then be tuned differently/separately depending on where the stealing occurs, e.g., [18]. Thus, this kind of approaches implicitly implies structuring parallel resources in some specific way. Most of previous works only report computational results in relatively small scales, i.e., not more than 256 cores, and for pure parallel benchmarks, e.g., UTS. Although those approaches are being designed in different distributed and computing contexts, they are complementary to ours in the sense that they could for instance be mixed and combined to deal with large scale hierarchical resources composed of several multi-core, multi-CPU, multi-cluster, multi-site, computing plate-forms, without changing the underlying algorithmic ideas or principles.

A crucial parameter common to all dynamic load-balancing schemes concerns the amount of work to be transferred from one node to another at runtime. Generally, when the sharable amount is very small, the large overhead is observed since many load balancing operations are performed. At the opposite, when it is very large, too few load balancing operations will occur, thereby resulting in large idle times when acquiring work despite the fact that surplus work is available. For instance, in [19], the authors made the observation that there may exist a fixed amount of work that results in the best performance. In fact, it is a critical parameter that can cause load imbalance if not tuned very carefully according to the application context. In [15], the steal-1 (one work unit at a time), steal-2 (two work units at a time) and steal-half (half of available work) strategies are analyzed. The authors argued that steal-half gives the best performance which is consistent with many other reported results, e.g., [20], [24]. To our knowledge, no previous work have explored the idea of adapting workload according to a logical overlay connecting computing resources.

Specific to load-balancing, our approach builds on previous studies by exploring the simple and intuitive idea that structuring networked resources and taking into account their induced logical computing power should allow them to efficiently self-coordinate their actions as proved in the context of peer-to-peer data-centric distributed systems. Such an approach is however not trivial to prove nor to effectively implement/experiment especially in large scale distributed message passing systems. In our study, we consider a simple tree overlay and adapt the load according to the size of induced subtrees, i.e., the bigger a subtree of a node is, the stronger computing power a node has. We did not take into account hardware issues, e.g., CPU, RAM, Network links, etc, though the computing ability and the mapping of hardware/network resources could be refined without changing the main design ideas.

2) Parallel B&B: While the load-balancing approach presented in this paper is well motivated from a pure parallel point of view, proving its effectiveness and its efficiency for Branch-and-Bound is to be considered as a contribution of independent interest. In fact, parallel B&B is more than a benchmark application with difficult combinatorial and computing challenges, e.g., see [5], [10], [12], [14], [21]. Due to lack of space, the reader is referred to [5], [12] for a more comprehensive and extensive discussion on the subject. Generally speaking, and besides specific bounding policies, special care has to be taken for the following tasks when parallelizing B&B: (i) efficient diffusion of best known bound, (ii) efficient work encoding to minimize message size and decrease communication latency, (iii) efficient work exploration and sharing policies. In [17], an extremely compact work encoding scheme for B&B is introduced. The authors therein also consider a master-worker model for parallelizing their B&B and successfully applied it to the Flowshop problem. In this paper, we use the same B&B work encoding proposed in [17], but our parallel exploration and work sharing policies are different. As will be argued later in our experimental study, the B&B Master-Worker scheme of [17] is extremely competitive and is found to even outperform the classical random work stealing approach. Among the hierarchical approaches dealing with B&B, the adaptive approach studied extensively in [2], [3] is the most recently and related to ours. That approach explicitly organizes masters in a tree like topology and adjust the load at each master according to its level in the hierarchy. In [17], the authors report 97% parallel efficiency of workers for their centralized architecture while solving a B&B instance generating extremely high coarse-grain works, i.e., the experiment was performed for a very big instance (Flowshop Ta_{56}): 22 years of sequential execution and 25 days of parallel execution. It is however not clear how that system would act when dealing with applications generating fine-grain works, which are known to be more challenging to schedule. Actually, the comparative study conducted in this paper shows that the approach of [17] leads to relatively good results at relatively small scales, but it can dramatically deteriorates for the higher scales. Besides, our generic approach is found to perform better both at the lower and higher scales, thus being accurate for balancing both fine and coarse grain works. In [2], [3], the authors report 99% of parallel efficiency *for their workers, with masters being excluded*. They also remarked that around 10% of deployed nodes were playing the role of masters. As discussed previously, the comparative study conducted in this paper reveals that substantial improvements are achieved by our approach.

D. Outline

The remainder of the paper is organized as follows. In Section II, we describe the high level design components of our overlay-centric load balancing approach. In Section III, we recall the UTS benchmark and discuss some B&B specific issues. In section IV, we present the results of our experiments. In Section V, we conclude the paper.

II. Description of our overlay-based approach

Let us assume that we are given an application which can recursively generate works without any initial knowledge about the number nor the complexity of work unit being processed. A work unit (or a task) in our terminology may (or may not) generate an unpredictable number of tasks at runtime. In this context, we describe the high level issues of our overlay-based approach. Because the lack of space, we only sketch some technical issues and do not give the detailed and technical message passing distributed protocols.

Generally speaking, logical overlays provide the property of structuring network resources in some specific way that helps the efficient execution of distributed tasks. This is typically the case for data centric peer-to-peer architectures where the logical overlay connecting available resources can be used at the aim of speeding up information query, routing tasks, etc. An overlay is in fact nothing else than a logical interconnection graph defining how nodes should cooperate locally to efficiently perform a given task. For high performance computing, overlays can also be viewed as basic tools to speed-up computations by enabling efficient work distribution.

In this paper, we basically structure the computing resources into a tree overlay. As will be argued later, a tree allows us to cope with basic distributed tasks, such as work distribution, and termination detection, in a simple and flexible manner without paying so much communication cost.

A. Basic work distribution in a structured tree overlay

Having packed the computing resources into a logical tree overlay, the application is initially pushed at the root node. Throughout the algorithm execution, an idle node first chooses a target peer among its neighbors to send a request asking for a piece of work. Actually, the strategy for peer selection plays a key role. In our distributed protocol, an idle node sends work requests downwards and upwards in the tree. In the down phase, every idle node first requests its children. Work requests are sent sequentially by choosing a child uniformly at random at each step. Then, if and only if all children are idle, a request is sent at last upwards to the parent. Notice that if a child has in parallel sent a work request to its parent, then the parent needs not to request that child, thus saving some communication steps in our protocol. Actually, this corresponds to a random work stealing strategy, but considering only the set of children which has not sent a request upwards yet.

This mechanism explores the locality induced by the overlay, as work inside a subtree will always be fully completed before getting more work from parent node. In addition, this allows us to easily cope with the difficult termination detection issue [7] without paying any further communication cost. In fact, the recursive nature of the distributed work request protocol guarantees that whenever a node receives a request from a child then all nodes in the sub-tree of that child have actually finished their work. Hence, when all the children of the root make a work request upwards to their root parent and the root itself has no more work to share, then the root can declare termination by sending a terminal signal to its children who further pass on the signal to their children and so on.

B. Cooperative overlay-dependent load-balancing

A challenging issue in dynamic load balancing is to avoid the situation where many nodes have few amount of work which is further split into fine grain units, while most of the work is being processed by few other nodes. This would in fact lead to the situation where many load balancing operations are performed, thus inducing a loss in parallel efficiency.

A tree overlay achieves network connectivity at the minimum communication cost. It tends to let workload flow from overloaded nodes to idle ones, thus naturally reducing imbalance caused by application irregularity while minimizing communication cost. However, one important limitation of a tree overlay is to force work to travel *only* across tree paths. In our approach, we consider to exploit the tree overlay into three different and complementary ways at the aim of reducing workload imbalance and speeding up the computations.

1) Tree degree/diameter: We study the issue of how the tree should be balanced for optimal performances. For that purpose, we consider to study different tree structures by essentially varying the degree of nodes. Intuitively, the higher the degree of a node is, the smaller diameter of a tree is, so it let work flows more quickly to achieve better efficiency. However, there is a threshold value, since if the degree of a node is beyond this value, the protocol is no longer scalable, e.g., Master-Worker or Star Topology.

2) Work sharing: We address the issue of what is amount of work to transfer when distributing the load. We consider the overlay-dependent strategy where a node divides its current work into the ratio of its own tree size and the tree size of the requesting node. Specifically, let u and v be two neighboring nodes with subtree size T_u and T_v, then either u is the children of v, or u is the parent of v. In the first case, the amount of work transferred from v to u is proportional to T_u/T_v. In the second case, it is in the ratio of $(T_u - T_v)/T_u$. One should notice that each node must know the size of its own subtree and also the size of its parent subtree. This is computed in a fully distributed manner using a classical converge-cast process starting from leaf nodes until reaching the root.

3) Bridge edges: We propose to extend the overlay tree to speed-up work flow from overloaded subtrees to under-loaded ones. For that purpose, we propose to connect nodes being far away each other in the tree using *bridges*. Those bridge edges are to be viewed as logical shortcuts that can be traveled by work to reach under-loaded subtrees more quickly. Actually, bridge edges tend to minimize the dependency of

our protocol on the tree diameter, thus leading to the best achievable performance of the overlay.

In our message passing distributed implementation, we allow every node v to *further* request work from *one* node u through a bridge edge $b_{v\to u}$ chosen at random. More precisely, *in parallel while requesting its neighbors in the tree*, every idle node v *asynchronously* sends a work request over $b_{v\to u}$. If u owns work, then it immediately services v with an amount of work in proportion to $T_v/(T_u + T_v)$. If u is idle, then this means that u has already sent an asynchronous work request through its bridge edge. Whenever an idle node, say r, gets work from its neighbors or through its bridge, then it services all nodes from which a work request was received. Let us remark that this distributed strategy operates in a recursive manner, implicitly building up a logical cluster of idle nodes. Consequently, all idle nodes are more likely to cooperate efficiently in searching for fresh work units.

From a more technical point of view, asynchronous work requests may cause deadlock issues, e.g., u chooses v as its $b_{u\to v}$, simultaneously v chooses u as its $b_{v\to u}$ and neither u nor v could get new works. Since the designed overlay assures that if there is a place which has work, there is always a path from it to other places in the system, so that this kind of issues can never happen in our approach. Notice also that a node could acquire more than one piece of work (from both a neighbor and a bridge), which we logically append to each other when computing the amount of work to send to other requesting nodes. Finally, extending the tree with bridge edges requires to slightly modify the termination detection mechanism. In fact, nodes always request their respective parents at last, but they have to distinguish the case where some work has been transferred asynchronously over a bridge edge. This is addressed by simply using aggregated work request messages without paying further communication cost.

III. Application Context

A. Unbalanced Tree Search

As stated previously, UTS [19] was designed to be the representative of dynamic applications producing highly irregular workload. It consists in the parallel exploration/counting of all nodes of a tree with extreme variation/imbalance in the relative size of its induced sub-trees, thus making UTS be an excellent adversary benchmark application for dynamic load balancing schemes. The reader is referred to [19] for further details on how UTS benchmark instances are generated.

B. Parallel Branch and Bound

B&B is a technique to find *optimal* solutions for hard combinatorial optimization problems coming from several application fields. It can be viewed as a divide and conquer algorithm, performing an implicit enumeration of the solution space corresponding to a given optimization problem. Generally speaking, B&B is similar to UTS since it consists in exploring the branches of an irregular/dynamic tree representing a solution space. However, B&B is more than a benchmark application with many challenging design issues, e.g., see [12], [14], [21]. In the following, we sketch the very general outline of our parallel B&B algorithm. For the sake of simplicity, we consider a permutation like combinatorial optimization problem of size s, where the goal is to find one permutation over $s!$ that maximizes (or minimizes) a given objective function. The B&B search process can be viewed as exploring a tree where the root represents the problem to be solved, a leaf represents a solution (a permutation) and an inner node represents a partial permutation, i.e., a sub-problem where only some variables of the permutation are fixed. We then use the B&B job encoding introduced in [17]. More precisely, the B&B tree is labeled in such a way, any subtree, corresponding to a sub-problem, can be uniquely encoded by an interval in $[0, s![$. Processing an interval consists in executing a sequential B&B for the sub-tree relative to the interval. In particular, we use the well-known algorithm proposed in [16] for bounding and a DFS traversal for branching. From a parallel point of view, the interval-based encoding is used as a compact and flexible data structure allowing us to parallelize the sequential B&B. In our implementation, we simply consider that the amount of work, which a node is processing, corresponds to the length of the interval. Then, a node can divide its interval into many pieces among idle neighbors at runtime following the discussed overlay dependent strategy. It is important to remark that the length of an interval is not representative of its relative complexity, since B&B can perfectly prune long intervals very quickly, and the opposite holds as well.

One crucial issue in parallel B&B is that a work (interval) should not be processed in a completely independent manner since the upper bound found during the search can impact the branching and the pruning when processing other works in parallel. Moreover, different work sharing strategies can lead to different upper bounds at runtime, thus making the performance of parallel B&B sensitive to the parallel exploration strategy. Thus, unlike UTS, the amount of work relative to a fixed B&B instance is sensitive to the way work is processed and shared. In our implementation, we run B&B from scratch without any specific upper bound. We further use a diffusing-like distributed protocol allowing each node to share at runtime the best found upper bound with its neighbors.

As benchmark application, we consider the NP-hard Flow-shop problem [25], which schedules a set of n jobs over m machines in order to minimize the total completion time, i.e., makespan. We consider the well-known Taillar's instances (Ta_{21}, $\cdots$, Ta_{30}) of the family Ta-$20*20$, i.e., 20 jobs and 20 machines [25]. Each instance of this family can require up to around 24 hours to be solved sequentially using one processor.

IV. Experimental Analysis and Results

All protocols were implemented using low level c++ libraries. Two clusters C_1 and C_2 of the Grid'5000 [13] were involved in our experiments. Cluster C_1 (resp. C_2) has 92 nodes (resp. 144 nodes), each one equipped with 2 CPU of 2.5 Ghz intel xeon processor with 4 cores per cpu (resp. 1 CPU of 2.6 Ghz intel xeon processor having 4 cores) and a network card infiniband-20G. Once some nodes of clusters C_1

n	Overlay		B&B				UTS			
			t_{avg}	σ	$t_{\max}$	$t_{\min}$	t_{avg}	σ	$t_{\max}$	$t_{\min}$
100	T_D	$d_{\max}=2$	**1742.9**	35.3	1787	1760	**2144**	247.14	2506	1863
		$d_{\max}=5$	**927.7**	91.07	1049	833	**1879.4**	47.76	1944	1813
		$d_{\max}=10$	**834.8**	27.8	870	799	**1756**	66.4	1861	1699
	T_R		**1412.4**	322.06	1629	774	**2162.2**	291.88	2555	1862
200	T_D	$d_{\max}=2$	**1231.5**	31.09	1265	1180	**1617.6**	239.6	1877	1278
		$d_{\max}=5$	**670.6**	33.06	721	612	**1232.8**	92.63	1325	1114
		$d_{\max}=10$	**462.2**	30.8	512	427	**1121.2**	64.3	1193	1021
	T_R		**1204.8**	253.45	1585	977	**1495.8**	136.4	1734	1389

TABLE I: Results using B&B Flowshop instance Ta_{21} and UTS with Binomial benchmark of size 157 billion nodes, i.e., generator parameters: (b=2000 q=0.4999995 m=2 r=599). t_{avg} (resp. σ, $t_{\max}$, $t_{\min}$) refers to the average (resp., standard deviation, maximum, minimum) execution time over 10 trials.

or C_2 are reserved through the Grid'5000 reservation system, they are exclusively owned by the user, but the network is not completely dedicated to that user. In the following experiments, all overlays are built before application execution, where each core is playing the role of a peer. No specific binding between peers and cores is adopted, i.e., peers are just thrown randomly on available cores. After completely built, the application (UTS or B&B) is pushed into an initial node, i.e., the root node in case of our approach, MW and AHMW, a random node in case of RWS, to start the parallel computation phase. For scale $n < 800$ cores, we use cores of cluster C_1. For scale $n \geq 800$, we use cores from both C_1 and C_2.

In the remainder, we use notation T_R to refer to our distributed protocol running over a *randomized tree* overlay constructed as following. Starting with the first node as root, each subsequent node chooses uniformly at random a node in the already constructed tree to be its parent. We use T_D to refer to a *deterministic tree* overlay with a fixed upper bound $d_{\max}$ on the maximum number of children per node. More precisely, depending on the number of peers, the overlay tree is constructed starting with a root node and packing at most $d_{\max}$ nodes in the first level. Then, we loop over the nodes of the new level packing again at most $d_{\max}$ children per node, and so on. We use BT_D to refer to the extended version of T_D (see Subsection II-B), where each node in T_D further chooses a *random bridge edge* to ask in parallel for work.

A. Property analysis of our approach

We start our experimental study by analyzing the impact of overlay properties and work sharing policies on performances. In our first set of experiments, we fix one benchmark instance for each of B&B and UTS and we study the performance of T_D and T_R under different overlay configurations. Results are summarized in Table I. We see that the execution time of the distributed protocol highly depends on the shape of the tree overlay. For a deterministic tree, execution time decreases as we increase the degree. Overall, a deterministic tree performs better compared to a randomized one (T_R). We also observe that as we increase $d_{\max}$, the protocol becomes more stable, i.e., the standard deviation σ decreases. In fact, as the degree of the tree increases, the distance between computing nodes decreases, thus making workload flows more quickly.

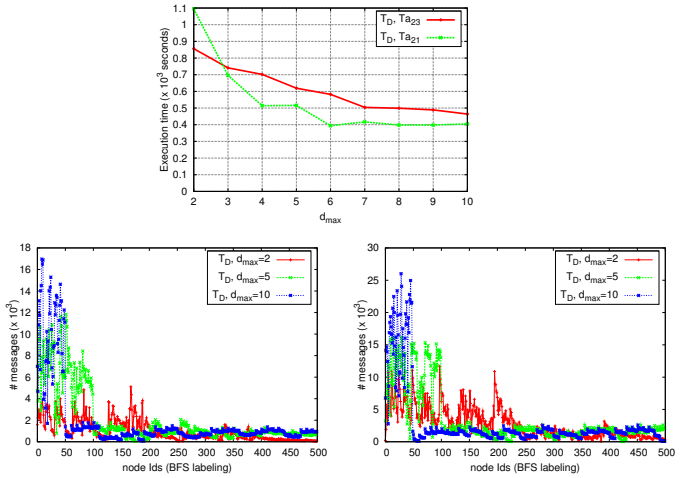

Fig. 1: **Top**: Execution time using 500 cores as a function of $d_{\max}$ for two B&B instances (Ta_{21} and Ta_{23}). **Bottom**: Number of messages sent by each peer respectively for instance Ta_{21} and Ta_{23}. The x-axis refers to node identifiers where nodes are numbered in a BFS manner, i.e., the root has id 0, nodes in the first level have ids 1 to 10, and so on.

To fully understand the impact of tree degree and diameter on execution time, load distribution and any congestion in the network, we conduct a second set of experiments at the higher scale of 500 cores. One can clearly see (Fig. 1 Top) that by increasing tree degree we gain in execution time, but quickly the gain becomes marginal as we increase the degree beyond some threshold (around 6). In fact, workload flows faster for large degree since the distance between nodes is minimized. However, this has a price, as confirmed by the distribution of message requests over tree nodes (Fig. 1 Bottom). Although execution time tends to decrease for larger degrees, communication load gets higher at intermediate nodes, i.e. message traffic is mostly supported by non-leaf tree nodes, thus inducing communication delays at those nodes.

In our third set of experiments, we focus on the performance of our strategy compared to the situation which the amount of transferred work is fixed by a parameter. More specifically, we consider the widely used strategy of dividing work in

two halves. We report execution time obtained for ten B&B instances at a scale of 200 cores and for UTS up to a scale of 128 cores in Fig. 2 Top Left and Bottom, respectively. One can clearly see that our subtree proportional work load distribution performs substantially better than the steal-half strategy, independently of B&B, UTS and network scale. In Fig. 2 Top Right, we draw the total number of work requests injected to the network by both strategies. We can clearly see that execution time and work requests are perfectly correlated. Thus, we can say that the overlay proportional strategy tends to guide the load balancing operations in order to result in the best performance. Recall that too few or too many load balancing operations both cause the imbalance situation from which the performance can fall down [19], [20].

We conclude this section by remarking that although the ten B&B Flowshop instances have the same theoretical size, their effective complexity may vary substantially making some instances harder to solve than the others. This can be attributed to two facts: (i) depending on the instance, B&B is able prune the search space more or less quickly, and (ii) work distribution implies starting exploring one region in the solution space before another one which can impact the best found solution upper bound, thus execution time.

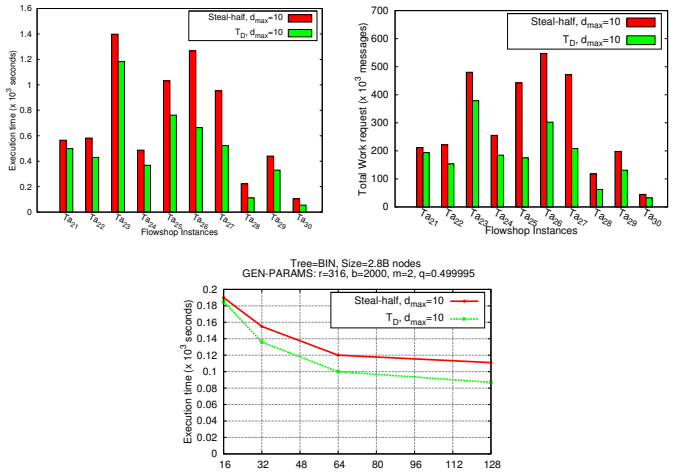

Fig. 2: Comparison with steal-half: **Top Left** (resp. **Right**) : Execution time (number of work requests) using 200 cores for the 10 B&B instances. **Bottom**: Execution time for UTS as a function of overlay size n.

B. T_D and BT_D vs Hierarchical Master-Worker for B&B

In this section, we compare our approach with the so-called adaptive hierarchical master-worker (AHMW) approach studied very recently in [2], [3], specifically for B&B. AHMW is mostly related to our work since it explicitly organizes computing nodes in a tree hierarchy, and then adapts the computations according to that tree. For the paper to be self-contained we recall the design principles and distributed policies used for AHMW. For further details on AHMW, the reader is referred to [2], [3]. Let us first notice that the *parallel B&B-specific search algorithm* induced by AHMW is conceptually different from our overlay approach in all aspects, i.e., work coding, work processing and work-sharing. In AHMW, computing nodes are organized in a hierarchical topology, thus inducing a tree backbone. Every node can both play the role of a master and/or a worker, depending on its height in the hierarchy. Furthermore, masters belonging to the same hierarchy level can directly communicate and share work with each other. The global B&B search tree is then decomposed into B&B subtrees which are mapped into the master hierarchy dynamically at runtime. In fact, the general idea of AHMW is to adapt the size of the B&B sub-trees being processed by each master/worker in an attempt to balance the load evenly. Roughly speaking, each master owns a work pool corresponding to sub-problems partially explored in its corresponding B&B sub-tree. When the work pool becomes empty, a master steals a sub-problem from its parent. It then re-generates a new work pool, and so on. B&B work grain plays a crucial role in AHMW. It corresponds to the depth at which a master/worker is allowed to explore a sub-problem. It is tuned to be a function of every master level in the overlay hierarchy which is shown to allow efficient and adaptive B&B work distribution among masters. (Notice that AHMW [2], [3] is argued to perform best when the tree hierarchy has degree 10, which is in a way consistent with our study). The results obtained with AHMW at scale of 200 cores in comparison with our approach for configurations $\mathrm{T_D}$, $\mathrm{BT_D}$ and $d_{max} = 10$, are summarized in Table. II.

	$\mathrm{T_D}$, $d_{\max} = 10$	$\mathrm{BT_D}$, $d_{\max} = 10$	*AHMW* [2], [3]
Ta_{21}	**499**	**354**	15804
Ta_{22}	**430**	**224**	438
Ta_{23}	1183	791	776
Ta_{24}	**368**	**194**	3352
Ta_{25}	**762**	**404**	2652
Ta_{26}	**664**	**472**	3231
Ta_{27}	523	**346**	445
Ta_{28}	**112**	**65**	1208
Ta_{29}	330	**68**	325
Ta_{30}	**55**	**29**	303

TABLE II: Execution time (in seconds) of T_D and BT_D compared with AHMW at scale of 200 cores. Bold style refers to execution time better than AHMW.

Using $\mathrm{T_D}$, we perform substantially better than AHMW for 7 out of 10 instances. Using $\mathrm{BT_D}$, our approach performs better than AHMW for 9 out of 10 instances. One can clearly see the relatively huge gap between AHMW and our approach, e.g., over all instances, $\mathrm{BT_D}$ (resp. $\mathrm{T_D}$) is approximately 10 (resp. 5) times faster than AHMW. This set of experiments also shows that $\mathrm{BT_D}$ performs significantly better than $\mathrm{T_D}$. This is naturally attributed to the bridge edges which are fully playing their role of speeding up workflow through the tree.

C. BT_D vs Master Worker vs Random Work Stealing for B&B

From a combinatorial point of view, the B&B search induced by AHMW is conceptually different from the one induced by our work coding and work sharing approach. In addition, the load-balancing policy developed in AHMW

specifically for B&B cannot be applied with the B&B interval-based work encoding used in our approach. Thus, to fully appreciate the performance of our approach independently from any B&B specific technicalities, we further compare it to the Master Worker (MW) approach studied in [17] and the well-known Random Work Stealing (RWS). For completeness, the important implementation issues raised by these two approaches are sketched below.

Firstly, the B&B specific operations used in MW [17] are similar to our approach in all aspects. However, the load-balancing and work distribution operations are fundamentally different. In particular, they are tuned to take into account specific properties of B&B works. This makes the MW approach to be an interesting candidate for evaluating the performance of our *generic* load-balancing scheme. In MW, there is a unique master playing the role of managing a global work pool for workers. The work pool at the master consists of a set of unprocessed B&B intervals and their corresponding workers. Workers periodically communicate with their master in order to update those intervals which have already been completed, and to acquire fresh work whenever their local work pool becomes empty. Whenever a master is asked for work, it chooses a B&B interval having the largest length and splits it to two halves. The right half is then transferred to the requesting worker, and the worker owning the original interval is notified. Actually, this corresponds to an *asynchronous* steal-half approach which is tuned at the aim of minimizing the communication bottleneck around the master. However, because the master could assign works *which are not completely disjoint* to different workers, some kind of redundancy may appear when executing the B&B in parallel. This issue is well studied in [17] and was shown to have very negligible impact on overall performance, i.e., the redundancy reported in [17] is of only 0.39% (in terms of B&B explored nodes).

Secondly, in RWS, an idle node selects at random another node and tries to steal work from it. We consider the standard steal-half strategy when transferring work among nodes. Besides, a critical issue relative to RWS, is to distributively detect termination without disturbing non-idle nodes and without congesting the network. In our implementation of RWS, we use the standard tree based Dijisktra termination detection algorithm taken from previous work stealing studies [11], [15].

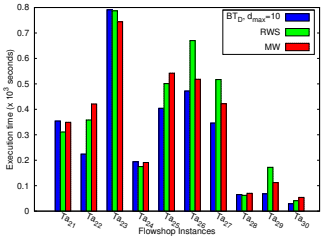

Fig. 3: Execution time of BT_D, RWS and MW for B&B instances at scale of 200 cores.

The performance of BT_D, MW and RWS for B&B is evaluated through a set of experiments at scale of 200 cores as reported in Fig. 3. Two conclusions can be withdrawn from this set of experiments. On one side, BT_D performs essentially better than MW and RWS. More precisely, BT_D outperforms on 7 out of 10 instances (Over all instances, BT_D is able to improve performance by up to 16% and 22% when compared to MW and RWS, respectively). On the other side, one can also see that the relative performance of the three approaches varies depending on the considered instance. Particularly, although the MW may seem rather simplistic at a first side, it is actually very competitive against RWS which is a reference approach for dynamic load balancing. This can be attributed to two facts: (i) The MW approach of [17] is well tuned to perform efficiently for B&B, and (ii) Such a centralized approach, which all dependencies are concentrated at a single point (the master), works well at a relatively low network scale. In next section, the relative scalability of the three approaches is analyzed in detail for both B&B and UTS.

D. Scalability of BT_D vs MW vs RWS for B&B and UTS

We start studying the relative scalability of our approach for parallel B&B and then we conclude with UTS. For B&B, we consider instance Ta_{21} and Ta_{23}, for which we found that MW and RWS perform better than our BT_D scheme at scale 200 cores. Then, we push our experiment further by scaling up to 1000 cores. As it can be observed in Fig. 4, the performance of MW starts to slow down while scaling up. Specifically, when using more than 600 cores, the execution time of Ta_{21} starts to increase rapidly and the execution time of Ta_{23} decreases very marginally. This is attributed to the severe communication bottleneck at the master caused by fine-grain works. This contrasts with our BT_D scheme which is fully distributed so that it continues scaling for both Ta_{21} and Ta_{23} while efficiently distributing communication load with fine-grain works.

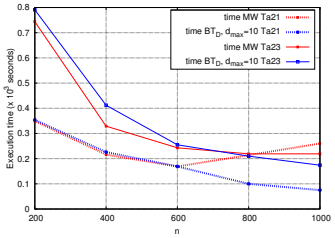

Fig. 4: Execution time of BT_D vs MW as a function of overlay size n for two critical instances Ta_{21} and Ta_{23}.

In Fig. 5 Top, we report the execution time and the corresponding parallel efficiency of BT_D against RWS for the two instances Ta_{21} and Ta_{23}. One can clearly see that RWS stays competitive up to 400 cores, but then it deteriorates dramatically compared with BT_D. Specifically, while the parallel efficiency of BT_D decreases marginally and stays above 90% (resp. 96%) for Ta_{21} (resp. Ta_{23}) in the scale of 1000 cores, it drops down quickly for RWS reaching 52% (resp. 63%) for Ta_{21} (resp. Ta_{23}). The relative scalability of BT_D is confirmed when executed for the UTS benchmark as shown in Fig. 5

Bottom. The parallel efficiency of BT_D is in fact substantially better than RWS, i.e, 77% vs 64%, using 512 cores.

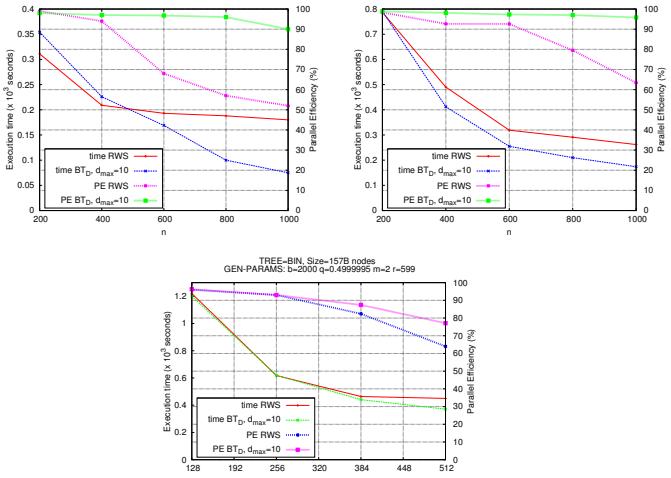

Fig. 5: BT_D vs RWS. Execution time and Parallel efficiency (PE), as a function of overlay size n, for B&B instances Ta_{21} (**Top Left**), Ta_{23} (**Top Right**), and UTS (**Bottom**)

Several conclusions can be drawn from this last set of experiments. For relatively low network scales, RWS is confirmed to be very competitive which is consistent with previous studies. However, there is still an opportunity for further improvements as demonstrated by our BT_D scheme which carefully explores the tree overlay properties. In fact, by simply extending tree paths with bridge edges, we allow work to flow more quickly. The load is then balanced more efficiently improving on RWS. At larger scales, RWS reaches its limits, since idle nodes try to catch victims 'blindly' using random requests. RWS can in fact be considered as operating over a fully connected overlay which implies communication overheads to find work. In contrast, our overlay centric approach tends to minimize communication delays by distributing the load in a more deterministic/cooperative manner and the gain in parallel efficiency, thus in speed-up, becomes substantial as we scale up the network.

V. Conclusion and Future Work

In this paper, we adopt an overlay-centric distributed strategy for dynamic load balancing in order to counteract the unpredictable nature of work induced by highly irregular applications. We study the properties of our approach compared to three state-of-the art approaches by conducting extensive real and large scale experiments using parallel B&B and UTS. In particular, by tightly coupling transferred workload and overlay properties, we are able to improve on previous results while scaling up to 1000 cores. As future work, we are aiming at designing new overlay based load-balancing strategies specific to large scale heterogenous distributed environments. In fact, we believe that mapping computing nodes into an overlay structure which is specifically adapted according to the nature of computing resources (e.g., multi-core, clusters, GPU, etc) would greatly help in guiding distributed load-balancing protocols through high performance and scalability.

References

[1] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. Atlas: an infrastructure for global computing. In 7^{th} *ACM SIGOPS Workshop on Systems support for worldwide applications*, pages 165–172, 1996.

[2] A. Bendjoudi, N. Melab, and E-G. Talbi. An adaptive hierarchical master-worker framework for grids: Application to B&B algorithms. *J. Parallel Distrib. Comput (JPDC)*, 72(2):120–131, 2012.

[3] A. Bendjoudi, N. Melab, and E-G. Talbi. Hierarchical branch and bound algorithm for computational grids. *Future Generation Computer Systems (FGCS)*, To appear, 2012. doi:10.1016/j.future.2012.03.001.

[4] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, 1999.

[5] Teodor Gabriel Craignic. Parallel branch and bound algorithms. In *Parallel Combinatorial Optimization (Chapter 1)*. Wiley, 2006.

[6] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, 1989.

[7] E. W. Dijkstra. Derivation of a termination detection algorithm for distributed computations. *Book Control Flow and Data Flow: concepts of distributed programming*, pages 507–512, 1987.

[8] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. Dynamic load balancing of unbalanced computations using message passing. In 21^{th} *Inter. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[9] Matteo F, Charles E. L, and Keith H. R. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, 1998.

[10] Raphael Finkel and Udi Manber. DIB - a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9:235–256, 1987.

[11] N. Francez and M. Rodeh. Achieving distributed termination without freezing. *IEEE Trans. Softw. Eng.*, 8:287–292, 1982.

[12] B. Gendron and T.G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[13] Grid500 French national gird. https://www.grid5000.fr/.

[14] Adriana Iamnitchi and Ian Foster. A problem-specific fault-tolerance mechanism for asynchronous, distributed systems. In *Inter. Conf. on Para. Proc. (ICPP)*, pages 4–13, 2000.

[15] D. James, L. D. Brian, P. Sadayappan, S. Krishnamoorthy, and N. Jarek. Scalable work stealing. In *ACM Inter. Conf. on High Performance Computing Networking, Storage and Analysis*, pages 53:1–53:11, 2009.

[16] J.K. Lenstra, B.J. Lageweg, and A.H.G.R. Kan. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[17] M. Mezmaz, N. Melab, and E-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. 21^{th} *Inter. Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, 2007.

[18] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical work stealing on manycore clusters. In 5^{th} *Conf. on Partitioned Global Address Space Prog. Models*, Oct 2011.

[19] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: an unbalanced tree search benchmark. In 19^{th} *inter. conf. on Languages and compilers for parallel computing (LCPC)*, pages 235–250, 2007.

[20] Stephen Olivier and Jan Prins. Scalable dynamic load balancing using upc. In 37^{th} *International Conference on Parallel Processing (ICPP)*, pages 123–131, 2008.

[21] J. F. Pekny and D. L. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. In *ACM conf. on Cooperation*, pages 56–62, 1990.

[22] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In 16^{th} *inter. conference on Parallel processing (Euro-Par)*, pages 217–229, 2010.

[23] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. Work stealing for multi-core hpc clusters. In 17^{th} *inter. conf. on Parallel processing (Euro-Par)*, pages 205–217, 2011.

[24] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. In 16^{th} *ACM Symp. on Principles and practice of parallel programming (PPoPP '11)*, pages 201–212, 2011.

[25] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

[26] R. V. Van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. *SIGPLAN Notices.*, 36:34–43, 2001.